

AD-A234 254

ENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: 20 Aug 1990 to 01 Mar 1993	
4. TITLE AND SUBTITLE SYSTEM KG, AlsyCOMP_025, Version 1.83, MIPS M/120-5 under RISC/os (Host & Targett), 90081411.11041				5. FUNDING NUMBERS	
6. AUTHOR(S) IABG-AVF Ottobrunn, Federal Republic of Germany					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IABG-AVF, Industrieanlagen-Betriebsgesellschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY				8. PERFORMING ORGANIZATION REPORT NUMBER IABG-VSR-074	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) SYSTEM KG, AlsyCOMP_025, Version 1.83, Ottobrunn, Germany, MIPS M/120-5 under RISC/os, Version 4.0 (Host & Target), ACVC 1.11.					
14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.				15. NUMBER OF PAGES	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		20. LIMITATION OF ABSTRACT	

AVF Control Number: IABG-VSR 074
20 August 1990

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #90081411.11041
SYSTEM KG
AlsyCOMP_025, Version 1.83
MIPS M/120-5 under RISC/os Host and Target

== based on TEMPLATE Version 90-04-03 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstrasse 20
W-8012 Ottobrunn
Germany

A-1

Certificate Information

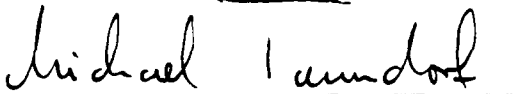
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 14 August 1990.

Compiler Name and Version: AlsyCOMP_025, Version 1.83

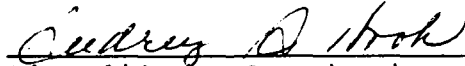
Host Computer System: MIPS M/120-5 under RISC/os, Version 4.0

Target Computer System: MIPS M/120-5 under RISC/os, Version 4.0

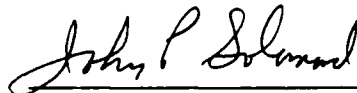
As a result of this validation effort, Validation Certificate #900814I1.11041 is awarded to Alsys. This certificate expires on 01 June 1992. This report has been reviewed and is approved.



IABG mbH, Abt. ITE
Michael Tonndorf
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany



Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311



Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE


Customer: System KG
Certificate Awardee: Alsys
Ada Validation Facility: IABG mbH. Abt. ITE
ACVC Version: 1.11

Ada Implementation:

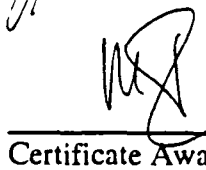
Ada Compiler Name and Version: AlsyComp 025, Version 1.83
Host Computer System: MIPS M/120-5 under RISC/os, Version 4.0
Target Computer System: MIPS M/120-5 under RISC/os, Version 4.0

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.


Customer Signature

Date: 17. August 1990


Certificate Awardee Signature

Date: 90 - August - 17

CONTENTS

CHAPTER	1	TEST INFORMATION	1
	1.1	USE OF THIS VALIDATION SUMMARY REPORT	1
	1.2	REFERENCES	2
	1.3	ACVC TEST CLASSES	2
	1.4	DEFINITION OF TERMS	3
CHAPTER	2	IMPLEMENTATION DEPENDENCIES	5
	2.1	WITHDRAWN TESTS	5
	2.2	INAPPLICABLE TESTS	5
	2.3	TEST MODIFICATIONS	8
CHAPTER	3	PROCESSING INFORMATION	10
	3.1	TESTING ENVIRONMENT	10
	3.2	TEST EXECUTION	11
APPENDIX	A	MACRO PARAMETERS	
APPENDIX	B	COMPILATION SYSTEM OPTIONS	
APPENDIX	C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro89] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro89]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro89] Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program
Office, May 1989.
- [UG89] Ada Compiler Validation Capability User's Guide, 24 October 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly

some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.

Ada Compiler Validation Capability (ACVC) The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.

Ada Implementation An Ada compiler with its host computer system and its target computer system.

Ada Validation Facility (AVF) The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.

Ada Validation Organization (AVO) The part of the certification body that provides technical guidance for operations of the Ada certification system.

Compliance of an Ada Implementation The ability of the implementation to pass an ACVC version.

Computer System Functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity Fulfillment by a product, process or service of all requirements specified.

Customer An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.

Declaration of Conformance A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.

Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro89].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 72 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 30 July 1990.

E28005C	B83022B	CB7001B	CD2A41E	CD7004C	CDA201E
B28006C	B83022H	CB7004A	CD2A87A	ED7005D	CE2107I
C34006D	B83025B	CC1223A	CD2B15C	CD7005E	CE2119B
B41308B	B83025D	BC1226A	BD3006A	AD7006A	CE2205B
C43004A	B83026B	CC1226B	CD4022A	CD7006E	CE2405A
C45114A	C83026A	BC3009B	CD4022D	AD7201A	CE3111C
C45346A	C83041A	AD1B08A	CD4024B	AD7201E	CE3118A
C45612B	B85001L	BD2A02A	CD4024C	CD7204B	CE3411B
C45651A	C97116A	CD2A21E	CD4024D	BD8002A	CE3412B
C46022A	C98003B	CD2A23E	CD4031A	BD8004C	CE3812A
B49008A	BA2011A	CD2A32A	CD4051D	CD9005A	CE3814A
A74006A	CB7001A	CD2A41A	CD5111A	CD9005B	CE3902B

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests) (*)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

C34007P and C34007S are expected to raise `CONSTRAINT_ERROR`. This implementation optimizes the code at compile time on lines 207 and 223 respectively, thus avoiding the operation which would raise `CONSTRAINT_ERROR` and so no exception is raised.

The following 21 tests check for the predefined type `LONG_INTEGER`:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35173D and B86001Z check for a predefined floating-point type other than `FLOAT`, `SHORT_FLOAT` or `LONG_FLOAT`.

C35702A, C35713B, C45423B, B86001T, C86006H check for a predefined type `SHORT_FLOAT`.

C41401A is expected to raise `CONSTRAINT_ERROR` for the evaluation of certain attributes, however, this implementation derives the values from the subtypes of the prefix at compile time as allowed by 11.6 (7) LRM. Therefore, elaboration of the prefix is not involved and `CONSTRAINT_ERROR` is not raised.

C45624A checks that the proper exception is raised if `machine_overflows` is false for floating point types with digits 5.

C45624B checks that the proper exception is raised if `machine_overflows` is false for floating point types with digits 6.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater.

C86001F recompiles package `SYSTEM`, making package `TEXT_IO`, and hence package `REPORT`, obsolete. For this implementation, the package `TEXT_IO` is dependent upon package `SYSTEM`.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

C96005B checks for values of type `DURATION'BASE` that are outside the range

of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type (see AI-00561).

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B is inapplicable as the collection size allocated is larger than the size specified by the 'STORAGE_SIZE' attribute.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

EE2401D contains instantiations of package DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR upon creation of such a file.

The 21 tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107C, CE2107D, CE2107L, and CE2108B attempt to associate names with temporary sequential files. The proper exception is raised when such an association is attempted.

CE2107H and CE2108D attempt to associate names with temporary direct files. The proper exception is raised when such an association is attempted.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external

file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B and CE3115A associate multiple internal text files with the same external file and attempt to read from one file what was written to the other, which is assumed to be immediately available. This implementation buffers output (see 2.3).

CE3112B attempts to associate names with temporary text files. The proper exception is raised when such an association is attempted.

CE3202A assumes that the NAME operation is supported for STANDARD_INPUT and STANDARD_OUTPUT. For this implementation the underlying operating system does not support the NAME operation for STANDARD_INPUT and STANDARD_OUTPUT. Thus the calls of the NAME operation for the standard files in this test raise USE_ERROR.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for external files. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 17 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

The following tests compile without error, as allowed by AI-00256 -- the units are illegal only with respect to units that they do not depend on. However, all errors are detected at link time. The AVO ruled that this is acceptable behavior.

BC3204C	BC3204D	BC3205C	BC3205D
---------	---------	---------	---------

IMPLEMENTATION DEPENDENCIES

CE3111B and CE3115A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests assume that output from one internal file is unbuffered and may be immediately read by another file that shares the same external file. This implementation raises END_ERROR on the attempt to read the file at lines 87 & 101, respectively (see 2.2).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact in Germany for technical information and sales information about this Ada implementation system, see:

System KG Dr. Winterstein
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

For a point of contact outside Germany for technical information and sales information about this Ada implementation system, see:

Alsys Inc.
67 South Bedford Str.
Burlington MA
01803-5152
USA
Tel. +617 270 0030

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 293 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic data cartridge containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation. Tests were compiled, linked and executed (as appropriate) using a single computer.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Tests were compiled using the command

```
sas.c 'file name'
```

and linked using the command

```
sas.link -o 'file name' 'main unit'.
```

The option -o was used to assign a dedicated file name to the generated executable image.

Chapter B tests, the executable not applicable tests, and the executable tests of class E were compiled using the full listing option -l. For several tests, completer listings were added and concatenated using the option -L 'file name'. The completer is described in Appendix B, compilation system options, chapter 4.2 of the User Manual on page 39.

Test output, compiler and linker listings, and job logs were captured on magnetic data cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The following macro parameters are defined in terms of the value V of \$MAX_IN_LEN which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

Macro Parameter	Macro Value
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
-----	-----
\$MAX_IN_LEN	255
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2147483647
\$DEFAULT_MEM_SIZE	2147483648
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	MIPS_UMIPS
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGUSR1)
\$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGUSR2)
\$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(SYSTEM.SIGALRM)
\$FIELD_LAST	512
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_NAME
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	0.0
\$GREATER_THAN_DURATION_BASE_LAST	200_000.0
\$GREATER_THAN_FLOAT_BASE_LAST	16#1.0#E+32
\$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.8#E+32
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0

MACRO PARAMETERS

```

$HIGH_PRIORITY      15

$ILLEGAL_EXTERNAL_FILE_NAME1
                    "/nodir/file1"
$ILLEGAL_EXTERNAL_FILE_NAME2
                    "/wrongdir/file2"

$INAPPROPRIATE_LINE_LENGTH
                    -1

$INAPPROPRIATE_PAGE_LENGTH
                    -1

$INCLUDE_PRAGMA1    PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2    PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST      -2147483648
$INTEGER_LAST       2147483647
$INTEGER_LAST_PLUS_1 2147483648

$INTERFACE_LANGUAGE C

$LESS_THAN_DURATION -0.0

$LESS_THAN_DURATION_BASE_FIRST
                    -200_000.0

$LINE_TERMINATOR    ASCII.LF

$LOW_PRIORITY       0

$MACHINE_CODE_STATEMENT
                    NULL;

$MACHINE_CODE_TYPE   NO_SUCH_TYPE

$MANTISSA_DOC        31

$MAX_DIGITS         15

$MAX_INT            2147483647
$MAX_INT_PLUS_1     2147483648
$MIN_INT            -2147483648

$NAME               SHORT_SHORT_INTEGER

$NAME_LIST          MIPS_UMIPS

```

MACRO PARAMETERS

\$NAME_SPECIFICATION1	/ben2/mp183/acvc11/chape/X2120A
\$NAME_SPECIFICATION2	/ben2/mp183/acvc11/chape/X2120B
\$NAME_SPECIFICATION3	/ben2/mp183/acvc11/chape/X3119A
\$NEG_BASED_INT	16#FFFFFFFE#
\$NEW_MEM_SIZE	2147483648
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	MIPS_UMIPS
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW INTEGER
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	10240
\$TICK	1.0/3600.0
\$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
\$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
\$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2
\$YOUR_PRAGMA	RESIDENT

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer (Chapter 4.1 of the User Manual, pages 35 ff). Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer (Chapter 5 of the User Manual, pages 50 ff). Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed.

§4.1 and Chapter 5 describe in detail how to call the Compiler, the Completer, which is called to generate code for instances of generic units, and the Linker.

Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.

The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.

Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

The command `sas.c` invokes the Compiler, and optionally Completer and Linker of the SYSTEAM Ada System.

<code>sas.c</code>	Command Description
--------------------	---------------------

NAME

`sas.c` – SYSTEAM Ada System compile command

SYNOPSIS

`sas.c` [option ...] [file ...] [-ld ldopt]

DESCRIPTION

Compilation, Completion and Linking are performed in that order. The Completer is called if the `-C` or the `-m` option is specified. The Linker is called if the `-m` option is specified. By default, only the compiler runs and compiles the source(s) in the given *files*.

The source file may contain a sequence of compilation units (cf. LRM(§10.1)). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The command delivers a non-zero status code on termination (cf. *exit*(2)) if one of the compilation units contained errors.

file specifies the file(s) to be compiled. The maximum length of lines in *file* is 255. The maximum number of source lines in *file* is 65534.

Note: If you specify a file name pattern, which is replaced by one or more file names by the shell, the order of the compilation is alphabetical, which is not always successful. Thus file name patterns should be used together with the option *-a*. With this option the sources can be processed in any order.

The generation of listing output is controlled by options *-l* and *-L*. The default listing filename for a compilation is the basename, cf. *basename*(1), of the source file with suffix *.l*; when the source file already has a suffix, it is replaced by the suffix *.l*. When an automatic recompilation is performed through option *-R* the basename is taken from the original source file name stored in the library.

-A Controls whether automatic inline expansion is performed. A sub-program *S* is automatically inlined at a place *P* where *S* is called, if the following conditions hold: *S* meets the requirements for explicit inlining via pragma *INLINE* (cf. §15.1.1). Spec and body of *S* are in the same compilation unit. The (estimated) size of the code of *S* is less than a fixed limit. If you specify *-A* automatic inline expansion is suppressed.

By default, automatic inline expansion is performed.

-a Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in *file* are entered into the library if they are syntactically correct. The actual compilation is done later.

Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.

By default, the normal, full compilation is done.

-C unitlist Requests the completion of the units in *unitlist*, which is a white space separated list of unit names. *unitlist* must be a single shell

argument and must therefore be quoted when it has more than one item. Example with two units:

```
sas.c -C "our_unit my_unit"
```

The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

If a listing is requested the default filename used is `complete.l`. The listing file contains the listing information for all units given in `unitlist`.

- c** Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the `sas.list -l` command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If `-c` is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.

- D** When linking, the generation of debug information is suppressed.
- I** Controls whether inline expansion is performed as requested by `PRAGMA inline`. If you specify `-I` these pragmas are ignored.

By default, inline expansion is performed.

- l** Generates listing files with default filenames (see above) in the current directory (use option `-L` for redirecting to another directory).
- L *directory*** Generates listing files with default filenames (see above) in *directory*.
- L *file*** Concatenates all listings onto file *file*.
- ld *ldopt*** This option can be used to supply options for the call of `ld` when linking a program by the `-m` option. `-ld` followed by the options to be passed to `ld(1)` must be the last items of the command.
- m *unit*** Specifies the name of a main program, which must be a parameterless procedure. This option will cause the completion of any generic instantiations in the program; if a listing is requested, the listing options have the same meaning as for the `complete` option; if the

completer has already been called by the `-C` option, the listing output is appended to that completer listing file. If all compilations are successful, the linker is invoked to build an executable program; if a listing is requested, the default filename for the linker listing is `link.l`.

- `-Ol` Restricts optimizations to level *l*. Level 0 indicates no optimizations, level 1 indicates partial optimizations, level 2 indicates full optimization. Default is full optimization. Partial optimization means those optimizations that do not move code globally. These are: constant propagation, copy propagation, algebraic simplifications, runtime check elimination, dead code elimination, peephole and pipeline optimizations. This Optimization level allows easier debugging while maintaining a reasonable code quality.
- `-o file` When linking is requested by `-m` this option can be used to specify the name of the generated executable program, which defaults to the unit name of the main program.
- `-R` Indicates that a recompilation of a previously analyzed source is to be performed. This option should only be used in commands produced by the `sas.make` command.
- `-r` Suppresses the generation of an executable object file when linking is requested. See the `-r` option of the `sas.link` command (§5) for details.
- `-S` Controls whether all run-time checks are suppressed. If you specify `-S` this is equivalent to the use of `PRAGMA suppress_all`.

By default, no run-time checks are suppressed, except in cases where `PRAGMA suppress_all` appears in the source.
- `-s` Controls whether machine code is appended to the listing file. `-s` has no effect if no listing is requested or `-a` (analyze only) is specified.

By default, no machine code is appended to the listing file.
- `-t` Suppresses selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable image. By default, the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable image.

Note: The code of the runtime system and of the predefined units is always linked selectively.

<code>-v</code>	Controls whether the <code>sas.c</code> command writes additional information onto standard error. By default, no additional information is written.
<code>-y library</code>	Specifies the program library the <code>sas.c</code> command works on. It needs write access to the library. The default library is <code>adalib</code> .

End of Command Description

4.2 Completing Generic Instances

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using the instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.

It is also possible to call the Completer explicitly with the `-C` option of the `sas.c` command.

4.3 Automatic Compilation

The SYSTEAM Ada System offers three different kinds of automatic compilation. It supports

- automatic recompilation of obsolete units
- automatic compilation of modified sources
- automatic compilation of new sources with unknown dependencies

In the following the term *recompilation* stands for the recompilation of an obsolete unit using the identical source which was used the last time. (This kind of recompilation could alternatively be implemented by using some appropriate intermediate representation of the obsolete unit.) This definition is stronger than that of the LRM (10.3). If a new version of the source of a unit is compiled we call it *compilation*, not a *recompilation*.

5 Linking

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The RISC/os system linker is used by the SYSTEAM Ada Linker.

To link a program, call the `sas.link` command. The Linker can also be called directly from the `sas.c` command and from the `sas.make` command.

<code>sas.link</code>	Command Description
-----------------------	---------------------

NAME

`sas.link` - invoke the SYSTEAM Ada System linker

SYNOPSIS

`sas.link` [option ...] unit [-ld ldopt]

DESCRIPTION

The `sas.link` command invokes the SYSTEAM Ada Linker.

The Linker builds an executable file. The default file name of the executable file is the unit name of the main program.

unit specifies the library unit which is the main program. This must be a parameterless library procedure.

-A Controls whether automatic inline expansion is performed. A subprogram *S* is automatically inlined at a place *P* where *S* is called, if the following conditions hold: *S* meets the requirements for explicit inlining via pragma `INLINE` (cf. §15.1.1). Spec and body of *S* are in the same compilation unit. The (estimated) size of the code of *S* is less than a fixed limit. If you specify `-A` automatic inline expansion is suppressed.

By default, automatic inline expansion is performed.

-c Suppresses invocation of the Completer of the SYSTEAM Ada System before the linking is performed. Only specify `-c` if you are sure

that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the `sas.link` command with different linker options.

-D By default debug information for the SYSTEAM Ada Debugger is generated and included in the executable file. When the `-D` option is present, debug information is not included in the executable file. If the program is to run under the control of the Debugger it must be linked without the `-D` option.

-I Controls whether inline expansion is performed as requested by `PRAGMA inline`. If you specify `-I` these pragmas are ignored.

By default, inline expansion is performed.

-l This option is passed to the implicitly invoked Completer, which by default generates a listing file `complete.l`. If `-l` is specified the Linker of the SYSTEAM Ada System creates a listing file containing a table of symbols which are used for linking the Ada units. This table is helpful when debugging an Ada program with the RISC/os debugger. The default name of the listing file is `link.l`. By default, the Linker does not create a listing file.

-L *directory* The listing files are created in directory *directory* instead of in the current directory (default).

-L *file* The listing files are concatenated onto file *file*.

-ld *ldopt* This option can be used to supply options for the call of `ld`. `-ld` followed by the options to be passed to `ld(1)` must be the last items of the command.

-ol This option is passed to the implicitly invoked Completer. See the same option with the `sas.c` command.

-o *file* Specifies the name of the executable file.
The default file name of the executable file is the unit name of the main program.

-r Suppresses the generation of an executable object file. In this case the generated object file contains the code of all compilation units written in Ada and of those object modules of the predefined language environment and of the Ada run time system which are used by the main program; references into the Standard C library remain unresolved. The generated object module is suitable for further `ld(1)` processing. The name of its entry point is `main`.

- S** This option is passed to the implicitly invoked Completer. See the same option with the `sas.c` command.
- s** This option is passed to the implicitly invoked Completer. See the same option with the `sas.c` command. If a listing is requested and `-s` is specified, the Linker of the SYSTEAM Ada System generates a listing with the machine code of the program starter in the file `link.1`. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.
By default, no machine code is generated.
- t** Suppresses selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable file. With `-t` the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable file.
- Note: The code of the runtime system and of the predefined units is always linked selectively, even if `-t` is specified.
- v** Controls whether the `sas.link` command writes additional information onto standard output.
By default, no additional information is written.
- y library** Specifies the program library the command works on. The `sas.link` command needs write access to the library unless `-c` is specified. If `-c` is specified the `sas.link` command needs only read access. The default library is `adalib`.

End of Command Description

The RISC/os System Linker is called with the command

```
/bsd43/bin/ld [-N ] -o resultfile /bsd43/usr/lib/cmplrs/cc/crt1.o \
    obj rtlib ld_options -lc /bsd43/usr/lib/cmplrs/cc/crtn.o
```

unless the `-r` option is specified. When `-r` is specified, the Linker is called with the command

```
/bsd43/bin/ld [-N] -o resultfile -r obj rtlib ld_options
```

Here, *obj* denotes the file containing the object module which is produced by the Ada Linker and *rtlib* the archive library containing the Ada runtime system. (This may be

`librtsdbg.a` resp. `librts.a` if the SYSTEAM Ada Linker is called with option `-D`. In this case the `-N` option is missing.)

If you invoke `ld` by yourself to link the executable object rather than having the Ada Linker doing it automatically, then you must explicitly specify a startup module (see below) and any libraries you want linked into the Ada program. Furthermore, the `ld` option `-N` should be specified to allow the resulting object file to be debugged by the SYSTEAM Ada System Debugger. (Note that debugging is only possible if the option `-D` was not passed to the Ada Linker.)

The startup module must satisfy the following requirements:

- A global variable called `environ` is defined containing a pointer to the current environment (cf. `exec(3-bsd)`).
- The Ada main program is called using the entry point `main`.
- `argc` and `argv` are passed as arguments to `main`.

Note that instructions following the call of `main` will never be executed.

By default, the Standard C startup routine `/bsd43/usr/lib/cmplrs/cc/crt1.o` is used.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. The package STANDARD is included in the implementer's Chapter 13, Predefined Language Environment, on pages 254ff.

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

PACKAGE standard IS

```

  TYPE boolean IS (false, true);
  -- The predefined relational operators for this type are as follows:
  -- FUNCTION "=" (left, right : boolean) RETURN boolean;
  -- FUNCTION "/=" (left, right : boolean) RETURN boolean;
  -- FUNCTION "<" (left, right : boolean) RETURN boolean;
  -- FUNCTION "<=" (left, right : boolean) RETURN boolean;
  -- FUNCTION ">" (left, right : boolean) RETURN boolean;
  -- FUNCTION ">=" (left, right : boolean) RETURN boolean;
  -- The predefined logical operators and the predefined logical
  -- negation operator are as follows:
  -- FUNCTION "AND" (left, right : boolean) RETURN boolean;
  -- FUNCTION "OR" (left, right : boolean) RETURN boolean;
  -- FUNCTION "XOR" (left, right : boolean) RETURN boolean;
  -- FUNCTION "NOT" (right : boolean) RETURN boolean;
  -- The universal type universal_integer is predefined.
  TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;
  -- The predefined operators for this type are as follows:
  -- FUNCTION "=" (left, right : integer) RETURN boolean;
  -- FUNCTION "/=" (left, right : integer) RETURN boolean;
  -- FUNCTION "<" (left, right : integer) RETURN boolean;
  -- FUNCTION "<=" (left, right : integer) RETURN boolean;
  -- FUNCTION ">" (left, right : integer) RETURN boolean;
  -- FUNCTION ">=" (left, right : integer) RETURN boolean;
  -- FUNCTION "+" (right : integer) RETURN integer;
  -- FUNCTION "-" (right : integer) RETURN integer;
  -- FUNCTION "ABS" (right : integer) RETURN integer;
  -- FUNCTION "+" (left, right : integer) RETURN integer;
  -- FUNCTION "-" (left, right : integer) RETURN integer;
  -- FUNCTION "*" (left, right : integer) RETURN integer;
```



```

-- FUNCTION "/" (left, right : integer) RETURN integer;
-- FUNCTION "REM" (left, right : integer) RETURN integer;
-- FUNCTION "MOD" (left, right : integer) RETURN integer;
-- FUNCTION "***" (left : integer; right : integer) RETURN integer;
-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end
-- with INTEGER as in SHORT_INTEGER or LONG_INTEGER. The
-- specification of each operator for the type universal_integer, or
-- for any additional predefined integer type, is obtained by
-- replacing INTEGER by the name of the type in the specification
-- of the corresponding operator of the type INTEGER, except for the
-- right operand of the exponentiating operator.
TYPE short_integer IS RANGE - 32_768 .. 32_767;
TYPE short_short_integer IS RANGE - 128 .. 127;
-- The universal type universal_real is predefined.
TYPE float IS DIGITS 6 RANGE
    - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;
FOR float'size USE 32;
-- The predefined operators for this type are as follows:
-- FUNCTION "=" (left, right : float) RETURN boolean;
-- FUNCTION "/=" (left, right : float) RETURN boolean;
-- FUNCTION "<" (left, right : float) RETURN boolean;
-- FUNCTION "<=" (left, right : float) RETURN boolean;
-- FUNCTION ">" (left, right : float) RETURN boolean;
-- FUNCTION ">=" (left, right : float) RETURN boolean;
-- FUNCTION "+" (right : float) RETURN float;
-- FUNCTION "-" (right : float) RETURN float;
-- FUNCTION "ABS" (right : float) RETURN float;
-- FUNCTION "+" (left, right : float) RETURN float;
-- FUNCTION "-" (left, right : float) RETURN float;
-- FUNCTION "*" (left, right : float) RETURN float;
-- FUNCTION "/" (left, right : float) RETURN float;
-- FUNCTION "***" (left : float; right : integer) RETURN float;
-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.
TYPE long_float IS DIGITS 15 RANGE
    - 16#0.FFFF_FFFF_FFFF_F8#E256 ..
      16#0.FFFF_FFFF_FFFF_F8#E256;
FOR long_float'size USE 64;
-- In addition, the following operators are predefined for universal
-- types:
-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)

```

```

        RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL;      right : UNIVERSAL_INTEGER)
        RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL;      right : UNIVERSAL_INTEGER)
        RETURN UNIVERSAL_REAL;
-- The type universal_fixed is predefined.
-- The only operators declared for this type are
-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
        right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
        right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- The following characters form the standard ASCII character set.
-- Character literals corresponding to control characters are not
-- identifiers.
TYPE character IS
    (nul,  soh,  stx,  etx,      eot,  enq,  ack,  bel,
     bs,   ht,   lf,   vt,      ff,   cr,   so,   si,
     dle,  dc1,  dc2,  dc3,      dc4,  nak,  syn,  etb,
     can,  em,   sub,  esc,      fs,   gs,   rs,   us,
     '!',  '"',  '#',  '$',     '%',  '&',  '...',
     '(',  ')',  '*',  '+',     '-',  '.',  '/',
     '0',  '1',  '2',  '3',     '4',  '5',  '6',  '7',
     '8',  '9',  ':',  ';',     '<',  '=',  '>',  '?',
     'A',  'B',  'C',  'D',     'E',  'F',  'G',
     'H',  'I',  'J',  'K',     'L',  'M',  'N',  'O',
     'P',  'Q',  'R',  'S',     'T',  'U',  'V',  'W',
     'X',  'Y',  'Z',  '[',     '\',  ']',  '^',  '_',
     'a',  'b',  'c',  'd',     'e',  'f',  'g',
     'h',  'i',  'j',  'k',     'l',  'm',  'n',  'o',
     'p',  'q',  'r',  's',     't',  'u',  'v',  'w',
     'x',  'y',  'z',  '{',     '|',  '}',  '~',  del);
FOR character USE -- 128 ascii CHARACTER SET WITHOUT HOLES
    (0, 1, 2, 3, 4, 5, ..., 125, 126, 127);
-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.
PACKAGE ascii IS
    -- Control characters:
    nul : CONSTANT character := nul;    soh : CONSTANT character := soh;
    stx : CONSTANT character := stx;    etx : CONSTANT character := etx;
    eot : CONSTANT character := eot;    enq : CONSTANT character := enq;
    ack : CONSTANT character := ack;    bel : CONSTANT character := bel;
    bs  : CONSTANT character := bs;     ht  : CONSTANT character := ht;
    lf  : CONSTANT character := lf;     vt  : CONSTANT character := vt;
    ff  : CONSTANT character := ff;     cr  : CONSTANT character := cr;
    so  : CONSTANT character := so;     si  : CONSTANT character := si;
    dle : CONSTANT character := dle;    dc1 : CONSTANT character := dc1;
    dc2 : CONSTANT character := dc2;    dc3 : CONSTANT character := dc3;

```

```

dc4 : CONSTANT character := dc4;   nak : CONSTANT character := nak;
syn : CONSTANT character := syn;   etb : CONSTANT character := etb;
can : CONSTANT character := can;   em  : CONSTANT character := em;
sub : CONSTANT character := sub;   esc : CONSTANT character := esc;
fs  : CONSTANT character := fs;    gs  : CONSTANT character := gs;
rs  : CONSTANT character := rs;    us  : CONSTANT character := us;
del : CONSTANT character := del;

-- Other characters:
exclam    : CONSTANT character := '!';
quotation : CONSTANT character := '"';
sharp     : CONSTANT character := '#';
dollar    : CONSTANT character := '$';
percent   : CONSTANT character := '%';
ampersand : CONSTANT character := '&';
colon     : CONSTANT character := ':';
semicolon : CONSTANT character := ';';
query     : CONSTANT character := '?';
at_sign   : CONSTANT character := '@';
l_bracket : CONSTANT character := '[';
back_slash : CONSTANT character := '\';
r_bracket : CONSTANT character := ']';
circumflex : CONSTANT character := '^';
underline : CONSTANT character := '_';
grave     : CONSTANT character := '`';
l_brace   : CONSTANT character := '{';
bar       : CONSTANT character := '|';
r_brace   : CONSTANT character := '}';
tilde     : CONSTANT character := '~';
lc_a      : CONSTANT character := 'a';
...
lc_z      : CONSTANT character := 'z';
END ascii;

-- Predefined subtypes:
SUBTYPE natural IS integer RANGE 0 .. integer'last;
SUBTYPE positive IS integer RANGE 1 .. integer'last;

-- Predefined string type:
TYPE string IS ARRAY(positive RANGE <>) OF character;
PRAGMA byte_pack(string);

-- The predefined operators for this type are as follows:
-- FUNCTION "=" (left, right : string) RETURN boolean;
-- FUNCTION "/=" (left, right : string) RETURN boolean;
-- FUNCTION "<" (left, right : string) RETURN boolean;
-- FUNCTION "<=" (left, right : string) RETURN boolean;
-- FUNCTION ">" (left, right : string) RETURN boolean;
-- FUNCTION ">=" (left, right : string) RETURN boolean;
-- FUNCTION "&" (left : string; right : string) RETURN string;
-- FUNCTION "&" (left : character; right : string) RETURN string;

```

```
-- FUNCTION "&" (left : string;    right : character) RETURN string;
-- FUNCTION "&" (left : character; right : character) RETURN string;
TYPE duration IS DELTA 2#1.0#E-14 RANGE
    - 131_072.0 .. 131_071.999_938_964_843_75;
-- The predefined operators for the type DURATION are the same
-- as for any fixed point type.
-- the predefined exceptions:
constraint_error : EXCEPTION;
numeric_error    : EXCEPTION;
program_error    : EXCEPTION;
storage_error    : EXCEPTION;
tasking_error    : EXCEPTION;
END standard;
```

13.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

- The package system
- The package calendar
- The generic procedure unchecked_deallocation
- The generic function unchecked_conversion
- The package io_exceptions
- The generic package sequential_io
- The generic package direct_io
- The package text_io
- The package low_level_io

13.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units

- The package collection_manager
- The package timing
- The package command_arguments
- The package text_io_extension

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED

has no effect.

ELABORATE

is fully implemented. The SYSTEAM Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit *u* was given, if the compiled unit contains an instantiation of *u* (or of a generic program unit in *u*) and if it is clear that *u* *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE

Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for ASSEMBLER and C. `PRAGMA interface(assembler,...)` provides an interface with the internal calling conventions of the SYSTEAM Ada System. See §15.1.3 for further description.

`PRAGMA interface(C,...)` is provided to support the C procedure calling standard. §15.1.4 describes how to use this pragma. The subprogram must not be a function returning an unconstrained array type. Nor must it have OUT or IN OUT parameters that are not passed by reference. If one of these restrictions is violated, the program is erroneous.

`PRAGMA interface` should always be used in connection with the `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

LIST

is fully implemented. Note that a listing is only generated when one of the listing options is specified with the `sas.c` (or `sas.make` or `sas.link`) command.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect; but see also the `-O` option with the `sas.c` command, §4.1.

PACK

see §16.1.

PAGE

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a ~ character in the listing.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of

the subtype priority, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package system (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if the PRAGMA priority 0 had been given (i.e. the task has the lowest priority).

SHARED

is fully supported.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see §15.1.2 for the implementation-defined PRAGMA `suppress_all`.

SYSTEM_NAME

has no effect.

15.1.2 Implementation-Defined Pragmas**BYTE_PACK**

see §16.1.

COMMENT `c_callable` not implemented in the umips version

EXTERNAL_NAME (`<string>`, `<ada_name>`)

`<ada_name>` specifies the name of a subprogram or of an object declared in a library package, `<string>` must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the call instruction for the subprogram. The subprogram declaration of `<ada_name>` must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.

Upper and lower cases are distinguished within `<string>`, i.e. `<string>` must be given exactly as it is to be used by external routines. The user should not define

external names beginning with an underline because Compiler generated names are prefixed with an underline. This pragma will be used in connection with the pragmas `interface (c)` or `interface (assembler)` (see §15.1.1) (see §15.1.2).

RESIDENT (<ada_name>)

this pragma causes the value of the object to be held in memory and prevents assignments of a value to the object <ada_name> from being eliminated by the optimizer (see §4.1) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : SYSTEM.address;
...
BEGIN
  x := 5;
  a := x'ADDRESS;
  do_something (a);  -- let do_something be a non-local
                    -- procedure
                    -- a.ALL will be read in the body
                    -- of do_something

  x := 6;
  ...
```

If this code sequence is compiled by the SYSTEAM Ada Compiler without the `-OO` option the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of `x` is not used before the next assignment to `x`. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of `x`.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

It will often be used in connection with the `PRAGMA interface (c, ...)` (see §15.1.4).

SUPPRESS_ALL

causes all the runtime checks described in the LRM (§11.7) to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the SYSTEAM Ada System, which are the same as those used for subprograms for which a PRAGMA interface (ASSEMBLER,...) is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada; it is provided in object form using the -ld option with the sas.link (or sas.c or sas.make) command.

In many cases it is more convenient to follow the C procedure calling standard. Therefore the SYSTEAM Ada System provides the PRAGMA interface(c,...), which supports the standard return of the function result and the standard register saving. This pragma is described in the next section.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The SYSTEAM Ada System uses three different parameter passing mechanisms, depending on the type of a parameter:

- *by value and/or result*: The value of the parameter itself is passed.
- *by reference*: The address of the parameter is passed (like an IN parameter of type system.address, which would be passed by value).
- *by descriptor*: A descriptor for the parameter is allocated at the callers side and is itself passed by reference.

The parameters of a subprogram are passed in registers where possible. The remaining parameters, if any, are passed in an area called *parameter block*. This area is aligned on a word boundary and contains parameter values (for parameter of scalar types), parameter addresses or descriptors addresses (for parameter of composite types) and alignment gaps.

For a function subprogram an extra register (\$r4 or \$f0) is assigned containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the JAL instruction. The address of the parameter block is passed in \$r3, if necessary. The static link of a subprograms is passed in \$r2, if necessary.

When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following: Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter register or into the parameter block before the call. Then, after the sub-program has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter register or the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, halfword or word boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a word boundary.

Parameters of composite types are passed by reference or by descriptor. The descriptors are allocated by the caller and are themselves passed by reference. A descriptor contains the address of the actual parameter object and further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.
- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint. This descriptor is itself passed by reference.
- For functions whose return value is an unconstrained array type a reference to a descriptor for the array is passed in the parameter block as for parameters of mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.
- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record.
If the parameter has mode OUT or IN OUT, the value of the CONSTRAINED attribute applied to the actual parameter is passed as an additional boolean IN parameter (which, when not passed in a register, occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and

the address are treated like two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a word boundary.

Ordering of parameters:

The ordering of the parameters is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. The registers \$r4..\$r22, \$r24..\$r25 and \$f0..\$f31 are available for parameter passing. A parameter block is only used when there are more parameters than registers of the appropriate class. Registers are used from low numbers to high numbers, the parameter block starts at offset zero and grows to higher offsets. Each parameter is handled as follows:

- A `short_float` parameter is allocated the next free even numbered floating point register (the corresponding odd numbered floating point register is unused for parameter passing). If no more floating point register is available one word is allocated in the parameter block, see below.
- A `float` parameter is allocated the next free floating point register pair. If no more floating point register is available a double word is allocated in the parameter block, see below.
- All other parameters (or their [descriptor] addresses, respectively) are allocated the next free general purpose register from \$r4..\$r22, \$r24..\$r25. If no more general purpose register is available for parameter passing, space is allocated in the parameter block depending on the representation of the parameter type, see below.
- If a parameter cannot be passed in a register, space is allocated in the parameter block as follows:

Because of the size and alignment requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. , In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter

instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, halfword or word boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter register or using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter register or parameter block. For a parameter of a composite type passed by reference the actual parameter value is accessed indirectly via the parameter address passed in a parameter register or in the parameter block. For a parameter of a composite type passed by descriptor the actual parameter value is accessed via the descriptor whose address is passed in a parameter register or in the parameter block. The descriptor contains a pointer to the actual object. When standard entry code sequences are used within the assembler subprogram (see below), the parameter block address is accessible at address -12(\$r30).

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, halfword, word or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs `pack` or `byte_pack` (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor that itself is passed by reference.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex record component clauses should be introduced for each record component if an object of that type is to be passed to a non Ada subprogram to be sure to access the components correctly.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers \$r1..\$r22, \$r24..\$r25 will be destroyed by the called subprogram and saves them of its own accord. The stack pointer \$r29 will have the same value after the call as before except for functions returning unconstrained arrays. The stack limit register (\$r24) will have the same value after the call as before unless the stack of the main task was extended. If the called subprogram wants to modify further registers it has to ensure that the old values are restored upon return from the subprogram. Note that these register saving conventions differ from the C calling standard.

Finally we give the appropriate code sequences for the subprogram entry and for the return, which both obey the rules stated above.

A subprogram for which PRAGMA interface(assembler,...) is specified is - in effect - called with the subprogram calling sequence

```

move    $4,...      | assign IN parameters, if any
move    $f0,...
...
jal     <subprogram address>
...
move    ....,$4      | read OUT parameters, if any
...
```

Thus the appropriate entry code sequence is

```

addiu   $sp,$sp,-12
sw      $0,0($sp)
sw      $fp,4($sp)
sw      $31,8($sp)
addiu   $15,$sp,-<frame_size-4>
```

```

    addiu    $fp,$sp,4
    slt      $1,$23,$15
    bne      $1,$0,L1
    nop
    jal      _EXTSTCK      | Storage check
    move      $24,$15
L1:
    move      $sp,$15
                                | The field at address -4($fp) is reserved
                                | for use by the Ada runtime system

```

The return code sequence is then

```

    move      $sp,$fp
    lw        $31,4($sp)
    lw        $fp,0($sp)
    jr        $31
    addiu     $sp,$sp,8

```

15.1.4 Pragma Interface(C,...)

The SYSTEAM Ada System supports PRAGMA interface(C,...).

With the help of this pragma *and* by obeying some rules (described below) subprograms can be called which follow the C procedure calling standard. As the user must know something about the internal calling conventions of the SYSTEAM Ada System we recommend reading §15.1.3 before reading this section and before using PRAGMA interface(C,...).

For each Ada subprogram for which

```
PRAGMA interface (C, <ada_name>);
```

is specified, a routine implementing the body of the subprogram <ada_name> must be provided, written in any language that obeys the C calling conventions cf. UMIPS Documentation Set, Languages Programmer's Guide, Chapter 3, in particular:

- Saving registers
- Calling mechanism
- C stack frame format.

RISC/os system calls or subroutines are allowed too.

The following parameter and result types are supported:

C Type	Ada Type
int	standard.integer
float	standard.float
double	standard.long_float
pointer	system.address

The calling mechanism for all parameter types is call by value. The type `address` may serve to implement all kinds of call by references: The user may build all kinds of objects and pass their addresses to the C subprogram or RISC/os system routine.

If

```
PRAGMA interface (assembler, <ada_name>);
```

is specified, a routine implementing the body of the subprogram `<ada_name>` and obeying the internal calling conventions of the SYSTEAM Ada Compiler must be provided.

The name of the routine which implements the subprogram `<ada_name>` should be specified using the pragma `external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not define names beginning with an underline.

The following example shows the intended usage of the pragma `interface` (C) to call a RISC/os system routine. The given procedure serves to open a file with a fixed name. It is called in the body of the main program.

```
WITH system;
PROCEDURE unix_call IS
  read_mode : CONSTANT integer := 8#0#;
  file_name : CONSTANT string := "/beni/test/f1" & ascii.nul;
  PRAGMA resident (file_name);
  ret_code : integer;
  use_error : EXCEPTION;
  FUNCTION unix_open (path : system.address;
                     oflag : integer) RETURN integer;
  PRAGMA interface (C, unix_open);
```



```
PRAGMA external_name ("open", unix_open);  
BEGIN  
  ret_code := unix_open (file_name'address, read_mode);  
  IF ret_code = -1 THEN  
    RAISE use_error;  
  END IF;  
END unix_call;
```

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.

If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.

If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM(§3.5.5(11))) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM(Annex C(13))) by the corresponding upper-case character. For example, `character'image(nul) = "NUL"`.

MACHINE_OVERFLOWS

Yields true for each real type or subtype.

MACHINE_ROUND

Yields true for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (STORAGE_SIZE, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. no length specification by STORAGE_SIZE given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed. If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:

If a length specification (STORAGE_SIZE, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM (§13.7) is reprinted here with all implementation-dependent characteristics and extensions filled in.

PACKAGE system IS

```
TYPE designated_by_address IS LIMITED PRIVATE;
TYPE address IS ACCESS designated_by_address;
FOR address'size USE 32;
FOR address'storage_size USE 0;
```

```
address_zero : CONSTANT address := NULL;
```

```
FUNCTION "+" (left : address; right : integer) RETURN address;
```

```
FUNCTION "+" (left : integer; right : address) RETURN address;
```

```
FUNCTION "-" (left : address; right : integer) RETURN address;
```

```
FUNCTION "-" (left : address; right : address) RETURN integer;
```

```
FUNCTION symbolic_address (symbol : string) RETURN address;
```

```
SUBTYPE external_address IS STRING;
```

```
-- External addresses use hexadecimal notation with characters
-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
-- "7FFFFFFF"
-- "80000000"
-- "8" represents the same address as "00000008"
```

```
FUNCTION convert_address (addr : external_address) RETURN address;
-- convert_address raises CONSTRAINT_ERROR if the external address

-- addr is the empty string, contains characters other than
-- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address value
-- cannot be represented with 32 bits.
```

```
FUNCTION convert_address (addr : address) RETURN external_address;
-- The resulting external address consists of exactly 8 characters

-- '0'..'9', 'A'..'F'.
```

```
TYPE name IS (mips_umips);
system_name : CONSTANT name := mips_umips;

storage_unit : CONSTANT := 8;
memory_size : CONSTANT := 2 ** 31;
```

```
min_int      : CONSTANT := - 2 ** 31;
max_int      : CONSTANT := 2 ** 31 - 1;
max_digits   : CONSTANT := 15;
max_mantissa : CONSTANT := 31;
fine_delta   : CONSTANT := 2.0 ** (-31);
tick         : CONSTANT := 1.0/3600.0;
```

SUBTYPE priority IS integer RANGE 0 .. 15;

TYPE interrupt_number IS RANGE 1 .. 32;

```
interrupt_vector : ARRAY (interrupt_number) OF address;
-- The mapping of signal numbers to interrupt addresses is
-- defined by this array.
```

```
sighup      : CONSTANT := 1;
sigint      : CONSTANT := 2;
sigquit     : CONSTANT := 3;
sigill      : CONSTANT := 4;
sigtrap     : CONSTANT := 5;
sigiot      : CONSTANT := 6;
sigabrt     : CONSTANT := sigiot;
sigemt      : CONSTANT := 7;
sigxcpu     : CONSTANT := sigemt;
sigfpe      : CONSTANT := 8;
sigkill     : CONSTANT := 9;
sigbus      : CONSTANT := 10;
sigsegv     : CONSTANT := 11;
sigsys      : CONSTANT := 12;
sigpipe     : CONSTANT := 13;
sigalrm     : CONSTANT := 14;
sigterm     : CONSTANT := 15;
sigusr1     : CONSTANT := 16;
sigusr2     : CONSTANT := 17;
sigchld     : CONSTANT := 18;
sigcld      : CONSTANT := sigchld;
sigcfz      : CONSTANT := 19;
sigstop     : CONSTANT := 20;
sigststp    : CONSTANT := 21;
sigpoll     : CONSTANT := 22;
sigio       : CONSTANT := 23;
sigurg      : CONSTANT := 24;
sigwinch    : CONSTANT := 25;
sigvtalrm   : CONSTANT := 26;
sigprof     : CONSTANT := 27;
```

```

sigcont    : CONSTANT := 28;
sigttin    : CONSTANT := 29;
sigttou    : CONSTANT := 30;
siglost    : CONSTANT := 31;

non_ada_error : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--   illegal instruction encountered
--   error during address translation
--   illegal address

TYPE exception_id IS NEW address;

no_exception_id      : CONSTANT exception_id := NULL;

FUNCTION constraint_error_id RETURN exception_id;
FUNCTION numeric_error_id   RETURN exception_id;
FUNCTION program_error_id   RETURN exception_id;
FUNCTION storage_error_id   RETURN exception_id;
FUNCTION tasking_error_id   RETURN exception_id;

FUNCTION non_ada_error_id   RETURN exception_id;

FUNCTION status_error_id    RETURN exception_id;
FUNCTION mode_error_id      RETURN exception_id;
FUNCTION name_error_id      RETURN exception_id;
FUNCTION use_error_id       RETURN exception_id;
FUNCTION device_error_id    RETURN exception_id;
FUNCTION end_error_id       RETURN exception_id;
FUNCTION data_error_id      RETURN exception_id;
FUNCTION layout_error_id    RETURN exception_id;

FUNCTION time_error_id      RETURN exception_id;

no_error_code        : CONSTANT := 0;

TYPE exception_information
  IS RECORD
    excp_id          : exception_id;
    -- Identification of the exception. The codings of
    -- the predefined exceptions are given above.
    code_addr        : address;
    -- Code address where the exception occurred. Depending
    -- on the kind of the exception it may be be address of

```

```

        -- the instruction which caused the exception, or it
        -- may be the address of the instruction which would
        -- have been executed if the exception had not occurred.
        error_code      : integer;
    END RECORD;

PROCEDURE get_exception_information
    (excp_info : OUT exception_information);
    -- The subprogram get_exception_information must only be called
    -- from within an exception handler BEFORE ANY OTHER EXCEPTION
    -- IS RAISED. It then returns the information record about the
    -- actually handled exception.
    -- Otherwise, its result is undefined.

PROCEDURE raise_exception_id
    (excp_id : exception_id);

PROCEDURE raise_exception_info
    (excp_info : exception_information);

    -- The subprogram raise_exception_id raises the exception
    -- given as parameter. It corresponds to the RAISE statement.

    -- The subprogram raise_exception_info raises the exception
    -- described by the information record supplied as parameter.
    -- In addition to the subprogram raise_exception_id it allows to
    -- explicitly define all components of
    -- the exception information record.

    -- IT IS INTENDED THAT BOTH SUBPROGRAMS ARE USED ONLY WHEN
    -- INTERFACING WITH THE OPERATING SYSTEM.

TYPE exit_code IS NEW integer;

error      : CONSTANT exit_code := 1;
success    : CONSTANT exit_code := 0;

errno : integer;
FOR errno USE AT symbolic_address ("errno");

PROCEDURE set_exit_code (val : exit_code);
    -- Specifies the exit code which is returned to the
    -- operating system if the Ada program terminates normally.
    -- The default exit code is 'success'. If the program is
    -- abandoned because of an exception, the exit code is
    -- 'error'.

```

PRIVATE

-- private declarations

END system;

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

target_type'SIZE > source_type'SIZE

15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM(Chapter 14) are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM(§13.10.1).

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation (release)` on a collection, all following calls of `release (unchecked deallocation)` until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM (§13.1), this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA PACK has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA PACK and is allowed at the same positions. For components whose type is an array or record type the PRAGMA BYTE_PACK has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA PACK all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA BYTE_PACK does not effect packing down to the bit level (for this see PRAGMA PACK).

16.2 Length Clauses

SIZE

for all integer, fixed point and enumeration types the value must be ≤ 32 ;
for `short_float` types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway);
for `float` types the value must be $= 64$ (this is the amount of storage which is associated with these types anyway).
for `long_float` types the value must be $= 96$ (this is the amount of storage which is associated with these types anyway);
for access types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a `RESTRICTION` error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 14). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

there is no implementation-dependent restriction. Any specification for `SMALL` that is allowed by the LRM can be given. In particular those values for `SMALL` are also supported which are not a power of two.

16.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type `integer` defined in package `standard`.

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a **RESTRICTION** error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a **RESTRICTION** error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and the difference between the sizes of the maximum and the minimum variant is greater than 32 bytes, and, in addition, if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object. (If the second condition is not fulfilled, the number of bits allocated for any object of the record type will be the value delivered by the size attribute applied to the record type.)
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM(§13.4(8))) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a **RESTRICTION** error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

16.5.1 Interrupts

Under RISC/os it is not possible to handle hardware interrupts directly within the Ada program; all hardware interrupts are handled by the operating system. In RISC/os, asynchronous events are dealt with by signals (cf. *sigvec(2)*). In the remainder of this section the terms *signal* and *interrupt* should be regarded as synonyms.

An address clause for an entry associates the entry with a signal. When a signal occurs, a signal catching handler, provided by the Ada runtime system, initiates the entry call.

By this mechanism, an interrupt acts as an entry call to that task; such an entry is called an *interrupt entry*. An interrupt causes the ACCEPT statement corresponding to the entry to be executed.

The interrupt is mapped to an *ordinary* entry call. The entry may also be called by an Ada entry call statement. However, it is assumed that when an interrupt occurs there is no entry call waiting in the entry queue. Otherwise, the program is erroneous and behaves in the following way:

- If an entry call stemming from an interrupt is already queued, this previous entry call is lost.
- The entry call stemming from the interrupt is inserted into the front of the entry queue, so that it is handled before any entry call stemming from an Ada entry call statement.

16.5.1.1 Association between Entry and Interrupt

The association between an entry and an interrupt is achieved via an interrupt number (type `system.interrupt_number`), the range of interrupt numbers being 1 .. 31 (this means that 31 single entries can act as interrupt entries). The meaning of the interrupt (signal) numbers is as defined in *sigvec(2)*. A single parameterless entry of a task can be associated with an interrupt by an address clause (the Compiler does not check these conventions). Since an address value must be given in the address clause, the interrupt number has to be converted into type `system.address`. The array `system.interrupt_vector` is provided for this purpose; it is indexed by an interrupt number to get the corresponding address.

The following example associates the entry `1r` with signal `SIGINT`.

```
...  
TASK handler IS  
  ENTRY ir;  
  FOR ir USE AT system.interrupt_vector (system.sigint);  
END;  
...
```

The task body contains ordinary accept statements for the entries.

16.5.1.2 Important Implementation Information

There are some important facts which the user of interrupt entries should know about the implementation. First of all, there are some signals which the user should not use within address clauses for entries. These signals are `sigfpe`, `sigsegv`, `sigbus` and `sigalrm`; they are used by the Ada Runtime System to implement exception handling and delay statements (`sigalrm`). Programs containing address clauses for entries with these interrupt numbers are erroneous.

In the absence of address clauses for entries, the Ada Runtime System establishes signal catching handlers only for the signals mentioned above, so all other signals will lead to program abortion as specified in the RISC/os documentation.

A signal catching handler for a specific signal is established when a task which has an interrupt entry for this signal is activated. The signal catching handler is deactivated and the previous handler is restored when the task has been completed. Several tasks with interrupt entries for the same signal may exist in parallel; in this case the signal catching handler is established when the first of these tasks is activated, and deactivated when the last of these tasks has been completed.

16.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

17 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

An external file is identified by a string that denotes a RISC/os file name. It may consist of up to 1023 characters.

The form string specified for external files is described in §17.2.1.1.

17.2 Sequential and Direct Files

Sequential and direct files are ordinary files which are interpreted to be formatted with records of fixed or variable length. Each element of the file is stored in one record.

In case of a fixed record length each file element has the same size, which may be specified by a form parameter (see §17.2.1.1); if none is specified, it is determined to be $(\text{element_type}'\text{SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$.

In contrast, if a variable record length is chosen, the size of each file element may be different. Each file element is written with its actual length. When reading a file element its size is determined as follows:

- If an object of the `element_type` has a size component (see §16.4) the element size is determined by first reading the corresponding size component from the file.
- If `element_type` is constrained, the size is the minimal number of bytes needed to hold a constrained object of that type.
- In all other cases, the size of the current file element is determined by the size of the variable given for reading.

17.2.1 File Management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in the LRM.

17.2.1.1 The NAME and FORM Parameters

The name parameter must be a RISC/os file name. The function name will return a path name string which is the complete file name of the file opened or created. Each component of the file name (separated by "/") is truncated to 255 characters. Upper and lower case letters within the file name string are distinguished.

The syntax of the form parameter string is defined by:

```
form_parameter ::= [ form_specification { , form_specification } ]
form_specification ::= keyword [ => value ]
keyword ::= identifier
value ::= identifier | numeric_literal
```

For identifier and numeric_literal see LRM(Appendix E). Only an integer literal is allowed as numeric_literal (see LRM(§2.4)). In an identifier or numeric_literal, upper and lower case letters are not distinguished.

In the following, the form specifications which are allowed for all files are described.

```
MODE => numeric_literal
```

This value specifies the access permission of an external file; it only has an effect in a create operation and is ignored in an open. Access rights can be specified for the owner of the file, the members of a group, and for all other users. numeric_literal has to be a three digit octal number.

The access permission is then interpreted as follows:

```
8#400#   read access by owner
8#200#   write access by owner
8#100#   execute access by owner
8#040#   read access by group
..       write/execute access by group, analogously
8#004#   read access by all others
..       write/execute access by others, analogously
```

Each combination of the values specified above is possible. The default value is 8#666#.

The definitive access permission is then determined by the RISC/os System. It will be the specified value for MODE, except that no access right prohibited by the process's

file mode creation mask (which may be set by the RISC/os `umask` command, cf. `sh(1)` and `umask(2)`) is granted. In other words, the value of each "digit" in the process's file mode creation mask is subtracted from the corresponding "digit" of the specified mode. For example, a file mode creation mask of `8#022#` removes group and others write permission (i.e. the default mode `8#666#` would become mode `8#644#`).

The following form specification is allowed for sequential, direct and text files:

`SYNCHRO => OFF | ON | ON_WAIT`

It allows reader/writer synchronization of parallel file accesses by different processes, such that only one process may write to a file (and no other process may read from or write to the same file in parallel) or multiple processes may read a file in parallel. This synchronization is achieved through the system call `flock(2)`.

By default parallel accesses are not synchronized (`SYNCHRO => OFF`).

If the form specification `SYNCHRO => ON` is given, `USE_ERROR` is raised when the access is not possible (because other processes are accessing the file when write access is requested, or because another process is writing the file when read access is requested).

If the form specification `SYNCHRO => ON_WAIT` is given, the process is blocked when the access is not possible for one of the above reasons. When the access becomes possible, the process is unblocked. `USE_ERROR` is not raised with `SYNCHRO => ON_WAIT`.

The following form specification is allowed for sequential and direct files:

`RECORD_SIZE => numeric_literal`

This value specifies the size of one element on the file (record size) in bytes. This form specification is only allowed for files with fixed record format. If the value is specified for an existing file it must agree with the value of the external file.

By default, $(\text{element.type}'SIZE + \text{system.storage_unit} - 1) / \text{system.storage_unit}$ will be chosen as record size, if the evaluation of this expression does not raise an exception. In this case, the attempt to create or open a file will raise `USE_ERROR`.

If a fixed record format is used, all objects written to a file which are shorter than the record size are filled up. The content of this extended record area is undefined. An attempt to write an element which is larger than the specified record size will result in the exception `use_error` being raised. This can only happen if the record size is specified explicitly.

17.2.1.2 Sequential Files

A sequential file is represented by an ordinary file that is interpreted to be formatted with either fixed-length or variable-length records (this may be specified by the form parameter).

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up. The content of this extended record area is undefined.

RECORD_FORMAT => VARIABLE | FIXED

This form specification is used to specify the record format. If the format is specified for an existing file it must agree with the format of the external file.

The default is variable record size. This means that each file element is written with its actual length. A read operation transfers exactly one file element with its actual length.

Fixed record size means that every record is written with the size specified as record size.

APPEND => FALSE | TRUE

If the form specification **APPEND => TRUE** is given for an existing file in an open for an output file, then the file pointer will be set to the end of the file after opening, i.e. the existing file is extended and not rewritten. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value **FALSE** is chosen.

TRUNCATE => FALSE | TRUE

If the form specification **TRUNCATE => TRUE** is given for an existing file in an open for an output file, then the file length is truncated to 0, i.e. the previous contents of the file are deleted. Otherwise the file is rewritten, i.e. if the amount of data written is less than the file size, data previously written will remain at the end of the file. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value **TRUE** is chosen.

The default form string for a sequential file is :

```
"RECORD_FORMAT => VARIABLE, APPEND => FALSE, " &
"TRUNCATE      => TRUE,      MODE   => 8#666# " &
"SYNCHRO      => OFF"
```

17.2.1.3 Direct Files

The implementation dependent type count defined in the package specification of `direct_io` has an upper bound of :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
```

A direct file is represented by an ordinary file that is interpreted to be formatted with records of fixed length. If not explicitly specified, the record size is equal to $(\text{element_type}'\text{SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$.

The default form string for a direct file is :

```
"RECORD_SIZE => ..., MODE => 8#666#, SYNCHRO => OFF"
```

17.3 Text Input-Output

Text files are sequential character files.

Each line of a text file consists of a sequence of characters terminated by a line terminator, i.e. an ASCII.LF character.

A page terminator is represented by an ASCII.FF character and is always preceded by a line terminator.

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last character of the file.

Output to a file and to a terminal differ in the following way: If the output refers to a terminal it is unbuffered, which means that each write request in an Ada program

will appear on the terminal immediately. Output to other files is buffered, i.e several characters are saved up and written as a block.

Terminal input is always processed in units of lines.

17.3.1 File Management

Besides the mode specification (cf. §17.2.1.1) the following form specification is allowed:

`APPEND => FALSE | TRUE`

If the form specification `APPEND => TRUE` is given for an existing file in an open for an output file, then the file pointer will be set to the end of the file after opening, i.e. the existing file is extended and not rewritten. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value `FALSE` is chosen.

The default form string for a text file is :

`"APPEND => FALSE, MODE => 8#666#, SYNCHRO => OFF"`

17.3.2 Default Input and Output Files

The standard input (resp. output) file is associated with the standard RISC/os files `stdin` resp. `stdout`.

Writing to the RISC/os standard error file `stderr` may be done by using the package `text_io_extension` (cf. §13.3.4).

17.3.3 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
FIELD'LAST = 512
```

17.4 Exceptions in Input-Output

For each of `name_error`, `use_error`, `device_error` and `data_error` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in LRM(§14.4).

NAME_ERROR

- in an open operation, if the specified file does not exist;
- if the name parameter in a call of the `create` or `open` procedure is not a legal RISC/os file name string; i.e, if a component of the path prefix is not a directory.

USE_ERROR

- whenever an error occurred during an operation of the underlying RISC/os system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a capacity limit is exceeded or for similar reasons;
- if the function name is applied to a temporary file or to the standard input or output file;
- if an attempt is made to write or read to/from a file with fixed record format a record which is larger than the record size determined when the file was opened (cf. §17.2.1.1); in general it is only guaranteed that a file which is created by an Ada program may be reopened and read successfully by another program if the file types and the form strings are the same;
- in a `create` or `open` operation for a file with fixed record format (direct file or sequential file with `form` parameter `RECORD_FORMAT => FIXED`) if no record size is specified and the evaluation of the size of the element type will raise an exception. (For example, if `direct_io` or `sequential_io` is instantiated with an unconstrained array type.)
- if a given `form` parameter string does not have the correct syntax or if a condition on an individual form specification described in §§17.2-3 is not fulfilled;
- in a `create` or `open` operation with form specification `SYNCHRO => ON` when the requested access is currently not possible; see §17.2.1.1 for the exact conditions.

DEVICE_ERROR

is never raised. Instead of this exception the exception `use_error` is raised whenever an error occurred during an operation of the underlying RISC/os system.

DATA_ERROR

the conditions under which `data_error` is raised by `text_io` are laid down in the LRM.

In general, the exception `data_error` is not usually raised by the procedure `read` of `sequential_io` and `direct_io` if the element read is not a legal value of the element type because there is no information about the file type or form strings specified when the file was created.

An illegal value may appear if the package `sequential_io` or `direct_io` was instantiated with a different `element_type` or if a different form parameter string was specified when creating the file. It may also appear if reading a file element is done with a constrained object and the constraint of the file element does not agree with the constraint of the object.

If the element on the file is not a legal value of the element type the effect of reading is undefined. An access to the object that holds the element after reading may cause a `constrained_error`, `storage_error` or `non_ada_error`.

17.5 Low Level Input-Output

We give here the specification of the package `low_level_io`:

```
PACKAGE low_level_io IS
  TYPE device_type IS (null_device);
  TYPE data_type IS
    RECORD
      NULL;
    END RECORD;
  PROCEDURE send_control    (device : device_type;
                             data    : IN OUT data_type);
  PROCEDURE receive_control (device : device_type;
                             data    : IN OUT data_type);
END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.

